TSDS

The Total Software Development System

Commodore 128 (128 mode) and Commodore 64

Includes:
Macro Assembler/Editor
Sprite Editor
Character/Matte Editor
Sound Effects Editor
Sample Source Code
and more...

Copyright 1985 by

NoSync

and

Kevin P. Pickell

Total Software Development System™
(Commodore 64/128 Version)

© 1985 by NoSync and Kevin P. Pickell
#103 2334 Marpole St
Port Coquitlam, BC V3C 2A2

Written by Kevin P. Pickell
January 1986 Release

The Total Software Development System™ consists of a Text Editor (SYSEDT™), Macro Assembler (SYSASM™), Sprite Editor (SPED™), Unassembler (UNASM™), MAE™ to TSDS™ converter, PAL™ to TSDS™ converter, Sound Effects Editor (SED™) and Matte/Character Set Editor (MACSED™).

Booting the System - C64

Insert the TSDS™ diskette into disk drive (Drive 0).

Type:

LOAD "0:*",8,1 (return)

The disk will operate for a while as TSDS™ is loaded.

Booting the System - C128

Insert the TSDS™ diskette into the disk drive.
Turn the Computer ON (Or OFF, then ON). TSDS Will load automatically.

The entire contents of this manual and all the corresponding software are
COPYRIGHT 1985 NoSync
ALL RIGHTS RESERVED

COPYRIGHT

This software is copyrighted and all rights reserved by NoSync. The distribution and sale of this product are intended for the use of the original purchaser only. Lawful users of the program are hereby licensed only to read the program from its medium into memory of a computer soley for the purpose of executing the program. Duplicating, selling, copying or otherwise distributing this product is a violation of the law.

This manual is copyrighted and all rights reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine language form without prior consent, in writing, from NoSync.

DISCLAIMER

NoSync MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THE PROGRAM DESCRIBED HEREIN, AS TO ITS QUALITY, PERFORMANCE, SALEABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. THIS PROGRAM IS SOLD "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE MEDIA PROVE DEFECTIVE FOLLOWING ITS PURCHASE, IT WILL BE REPLACED BY NoSync FOR A NOMINAL CHARGE. NoSync WILL ALSO SUPPLY, TO REGISTERED OWNERS OF THIS PROGRAM, UPON REQUEST, AND FOR A NOMINAL CHARGE, UPDATES TO THE PROGRAM.

Dedicated to Jim Baurle
for introducing me to
the wonderful world of computers.
Kevin P. Pickell.

OVERVIEW

TSDS is divided into two separate programs. The ASSEMBLER/EDITOR and the SPRITE EDITOR. The Editor/Assembler resides in memory from $7800-$BFFF. Memory from $0800-$77FF is used for user programs and source code files. During an assembly, memory from $0800-$77FF is used for symbol table and memory from $D000-$FFFF is used for macros and error files. The memory from $C000-$CFFF is always left unused.

The sprite editor resides in memory from $0800-$32FF. It uses from $3300-$3FFF for variables and $4000-$7FFF for sprites. Because the sprite area overlaps a portion of the Editor/Assembler, the editor/assembler must be disabled (KILL) if sprites 224-255 are edited because they will be stored in the program.

FAST LOAD ROUTINES

Fast load routines have been added to all of the LOADing within TSDS and Sprite/Editor. The code for the routines were written and implemented by Jeff Lydiatt (Author of Super Kwik Copy).

THANKS

Thanks to the following people for putting up with the preliminary versions while helping to debug and improve TSDS.

Rick Younie Jim Farewell Kris Hatlelid
ACCESS Software Electronic Arts
Glen Hazelwood Jeff Lydiatt Brian Niessen

And the gang from Sunday Software and the Commodore Computer Club, Vancouver.

TSDS Table of Contents

Extended DOS Commands (SUMMARY) A-1
Extended EDITOR Commands (SUMMARY) A-1

Printer Support A-2
DOS SUPPORT EXTENSIONS
>L A-2
>S A-3
>$ A-3

EDITOR COMMANDS
A-3
BLOAD/BRUN/LOAD/(uparrow) A-3
APPEND A-4
ASSEMBLER, ASM A-4
AUTO A-4
CDISK A-4
CHANGE A-5
COLOUR A-5
HELP A-5
CONV A-5
DELETE A-6
DFREE A-6
DISK A-6
EDIT A-6
EXEC A-6
FIND A-6
FVER A-7
GET A-7
KEY A-7
KEYS A-7
KEYC A-7
KEYL A-7
KEYR A-8
KILL A-8
LIST A-8
LOADAT A-8
LOG A-9
OLD A-9
PASSWORD A-9
PRINTER A-9
PUT/CPUT A-10
RENUMBER A-10
RESTORE A-10
FAST A-10
SLOW A-10

TSDS ASSEMBLER
Introduction B-1
An Assembly B-1
Source Line Format B-3
Labels B-4
Addressing Modes B-4
Comments B-5
Assembler Expressions B-5

Term Formats
Decimal B-6
Hexadecimal B-6
Octal B-6
Binary B-6
Program Counter B-6
Character B-6
Label B-6
Expression B-7
Negation B-7
Low/high Byte B-7
Forced Absolute Addressing B-7
Operators B-7

ASSEMBLER Directives
= C-1
(backarrow) C-2
*= C-2
*S= C-2
.END C-2
.BYT C-3
.ASC C-3
.POK C-3
.WOR C-3
.DBY C-3
.FIL C-4
.LIB C-4
.BIN C-4
.SST C-5
.IFx C-5
.ADD C-5
.EOR C-5
.SCS C-5
.CCS C-6
.LPP C-6
.PAG C-6
.SKI C-6
.LIS C-6
.NOL C-6
.SYM C-7
.SYN C-7
.SYO C-7
.NOS C-7
.INV C-7
.INB C-7
.PRT C-7
.MAC C-8
.MND C-8

.MEX	C-8
.MDX	C-8
MACRO CAPABILITIES	D-1
Example legal parameters	D-2
Macro Examples	D-2
ERROR MESSAGES	E-1
SPRITE EDITOR	
Editing Keys	F-1
Commands	F-1
ANIMATE	F-2
COLOR CHANGE MENU	F-2
BLOCK FILL	F-2
INTERLACE	F-2
JOYSTICK FIRE MODE	F-2
KEYBOARD AUTO CURSOR MOVE	F-2
LOAD	F-3
OVERLAY	F-3
OVERLAY MODE OFF	F-3
PUSH MENU	F-3
SAVE	F-3
TRANSFER	F-4
VIEW	F-4
UNASSEMBLER	G-1
CONVERTERS	
MAE/PAL to TSDS	G-2
SOUND EFFECTS EDITOR	H-1
MATTE/CHARACTER EDITOR	I-1
C128 CHANGES/ENHANCEMENTS	**J-1**

SYSEDT is an operating system which works in conjunction with the BASIC operating system in your C64 to create a better editing environment.

SYSEDT features include directory oriented commands, auto repeat key, up/down scrolling through source files and BASIC programs, extended DOS support commands, extended editor commands, and simplified output to printer.

Extended DOS Commands

>n	New Diskette
>i	Initialize diskette
>v	Validate diskette
>d	Backup diskette
>c	Copy files
>r	Rename a file
>s	Scratch a file
>$	List the directory
>l	List a file

Extended Editor Commands

#	Display version number and editor status
/	Load a file
uparrow	Load and RUN a file
BLOAD	Load a machine language program
BRUN	Load ML program and JMP to start address
APPEND	Append program to end of current program
ASM, ASSEMBLE	Assemble a source file
AUTO	Set auto line number
CDISK	Change a disk drive device number
CHANGE	Change one pattern to another pattern
COLOUR	Set the background and border colours
CONV	Numeric Base Conversion: Binary, Decimal, Hex, and ASCII
CPUT	Put a text file to disk and crunch spaces
DELETE	Delete a range of lines
DFREE	Display diskette name and blocks free
DISK	Select disk drive device number
EDIT (S,B)	Select source file/BASIC programs
EXEC	Execute a text file as keyboard commands
FIND	Find occurrences of a pattern
FVER	Verify current text against a disk file
GET	Read a text file into the editor
HELP	Displays all of the extended editor commands
KEY	Define a function key
KEYS[-]	Turn function keys on/off
KEYC	Clear all function keys
KEYL	List all function keys and definitions
KEYR	Set repeat key speeds
KILL	Turn off the editor
LOADAT	Load a program file at any address
LIST	Improved list command
LOG	Display start/end address of a program file
OLD	Restores a program after the 'NEW' command
PASSWORD	Password Protect Source Files

PRINTER	Select printer device number and linefeeds
PUT	Put a text file to disk
RENUMBER	Renumber all or part of a program
RESTORE	Restore all editor conditions to default
FAST	Turn on Fast DOS [Default on]
SLOW	Turn off Fast DOS.

DOS Support Extensions

One of the unique features of the Commodore computer systems is that the peripheral units are totally separate computers, allowing them to function independently, without supervision from the main computer. The programmer merely sends a command to the disk drive using Commodore's own DOS (Disk Operating System) language, and the disk drive interprets the command and performs the necessary steps to execute the command.

DOS Support

The primary function of the DOS Support Commands of SYSEDT is to allow access to the DOS language of the disk drive. This is done by typing one of the DOS support keys (**>**, **!**, **@**, or **backarrow**), followed by a valid DOS command. DOS commands are the commands where the disk drive manual tells you to open a file to device 8, secondary address 15, and print the command to that channel. The DOS support commands do not require you to open a channel. A DOS instruction may be enclosed in quotes to allow control characters to be included. SYSEDT allows you to use a calculated string expression in the DOS command message. The only restriction is that the message must begin with a string in quotes, even if it is a null string. **ie: >""+A$+CHR$(1)** If a DOS SUPPORT KEY is entered with nothing after it, the disk error channel is read, and the current disk message is displayed. Errors are only displayed the first time the error channel is read. Any subsequent attempts will return an '00,OK,00,00' message.

Printer Support

Preceeding any BASIC or SYSEDT command with an asterisk (*) will send output from that command to the selected printer device. Use the **PRINTER** command to select the printer device.

Command:>l
Function: List a program or file from the disk
Syntax: >l"filename"
Description: The file will be listed as a BASIC program if it is a PRG file, or as a source file if it is a SEQ file. The contents of the file will be listed to the screen or printer. The file is not loaded into memory, so it will not affect the current program. At any time during the file listing, output may be paused by tapping the SPACE BAR, pressing any key to continue. The output may be aborted by pressing the STOP key. The command *>l"filename" will send the listing to the current printer device.

Command:>s
Function: Scratch (delete) a file(s)
Syntax: >s [drive] :"filename"
Description:A file or group of files may be deleted by typing >s [drive]: in the margin beside the file you wish to remove from the directory. Full pattern matching may be used.

Examples
>s1: "xfil*"
>s0:a?1

Command:>$
Function: List the disk directory
Syntax: >$ [drive][:"pattern"]
Description: The >$ command displays the disk directory in exactly the same format is if it were loaded into memory, and then listed. All of the Commodore DOS pattern matching characters are permitted. The command *>$ will send the directory to the current printer device.

Extended Editor Commands

Command:#
Function: Display version and editor status
Description: The current version number of SYSEDT will be listed along with the current status of the following: Editor, Auto increment, Disk device number, Key Status, and the Printer device number.

Example
#
**** SYSEDT V1.x JANUARY 1986 (TSDS) ****
(c) 1985 BY NOSYNC and KEVIN P. PICKELL
EDIT:S:AUTO:N:DISK:8:KEY?:Y:PRNT:4

Command:/, uparrow, BLOAD, BRUN
Function: Load(/), Load and Run (uparrow), BLOAD, and BRUN
Syntax: /"filename"
BRUN filename
Description: These commands provide a quick and easy way to load a program from disk. If used from the directory, all numbers before the quotation mark will be ignored. The '/' command loads the file into memory in the BASIC program work space. The uparrow command is the same as the '/' command except that the program is run after it is loaded. The 'BLOAD' command is like the '/' command except that the file is loaded at its load address. If the uparrow command has no file name beside it, the current program in memory is RUN.

Command:APPEND
Function: Append a program from disk onto the program in memory.
Syntax: APPEND"filename"
Description: The append function takes a program or source file from the disk drive and adds it on to the end the program in memory.

Command:ASSEMBLE, ASM
Function: Assemble a source file from disk.
Syntax: ASM "sourcefile" [to "objectfile"]
Description: The assemble command runs the assembler portion of TSDS. If an assembly listing is required then use a * before the ASSEMBLE command. (The listing will be send to the current printer device.) If an object name is present then the assembly output is sent to the file. After the assembly type LIST for all of the assembly error messages. The ASSEMBLE command will erase any source file or program that may have been in memory when the command was executed.

Command:AUTO
Function: Set/Clear auto line numbering mode
Syntax: AUTO [<increment>]
Description: After a line is entered, the computer will generate the next line number based on the one you just entered. If you change the number, the computer will automatically pick up at that point. For example, if you entered 'AUTO10' and you then enter line 100, the computer would feed you line 110. If you now change that to line 150, the computer would then feed you with 160. To turn the auto function off, enter AUTO with no parameter.

Command:CDISK
Function: Change the device number of a disk drive (in software)
Syntax: CDISK <olddev> to <newdev>
Description: If more than one disk drive is connected to the same C-64, each drive must have its own unique device number. For example: If you have two 1541 drives that are both device number 8 and you wish to have them connected to the same C-64, then hook up one only, type CDISK 8 TO 9 and press (return). Then hook up the second drive to the C-64. Now the first drive is device 9 and the second drive is device 8. Resetting the disk drive(s) or computer will cause the drive(s) to reset and their device number(s) will return to their normal power up state.

Example
CDISK 8 TO 9
CDISK 9 TO 8

Command:CHANGE
Function: Change occurrences of a pattern to another pattern
Syntax: CHANGE[B][E][V][P][C][R]["] del oldpat del ["] del newpat del [,range]

OPTIONS

B	change only if at the Beginning of a line
E	change only if at the End of a line
V	change only if an exact Variable (or Label) match
P	enable pattern matching ('#' as a wild card)
C	change if only in command area-not in quotes or DATA
R	change Remainder of line to new pattern
"	turn quote mode on (disable crunching)

Description: The change command finds all occurrences of a pattern within the range of the program and changes them to another pattern, displaying the resulting lines. 'del' is any character that is not in 'oldpat' or 'newpat' or any of the 'option' characters. The options B, E, V, P, C, and R may be combined to give the desired selectivity.

Command:COLOUR
Function: Set the background and border colours
Syntax: COLOUR <background>, <border>
Description: The colour command sets the background and border colours to the values given.

Command:HELP
Function: Displays all of the Extended Editor Commands
Description: The help function displays all of the names of the extended editor commands.

Command:CONV
Function: Convert numbers between bases
Syntax: CONV <value>

implied= decimal
%= binary
@= octal
$= hex
"= ascii

Description: The CONV command accepts values in the above forms and then displays the same value in decimal, hex, binary and ASCII .

Example:
CONV $41
65 $0041 %0000 0000 0100 0001 "@A

Command:DELETE
Function: Delete a range of lines from the current program
Syntax: DELETE <range>
Description: The delete command removes an entire group of lines at one time. Although it seems reasonable that the DELETE command with no range should delete the entire program, a NEW command must be used in its place.

Command:DFREE
Function: To display the diskette name and blocks free
Description: The DFREE command is a quick way to see the specifications a diskette without having to look at all of the directory entries.

Command:DISK
Function: Set default disk device number
Syntax: DISK <device number>
Description: The disk command defines the default device number to be used for disk oriented commands. The purpose of this command is to allow multiple disk drive operation.

Command:EDIT
Function: Select editing mode (Source files or BASIC programs)
Syntax: EDIT [B][S]
Description: The SYSEDT editor can be used to edit either BASIC programs or source files. The default is source files. The command EDIT S puts SYSEDT in source mode and the command EDIT B puts SYSEDT in basic program mode.

Command:EXEC
Function: Execute a text file as keyboard commands
Syntax: EXEC "filename"
Description:EXEC is an extremely powerful command which allows you to prepare a long list of commands which will be automatically entered as if they were typed on the keyboard. EXEC cannot execute text files longer than 12k. Common uses of the EXEC command are to setup definable keys, set disk and printer device numbers, or anything else which needs to be done repeatedly.

Command:FIND
Function: Find occurrences of a pattern within a program
Syntax: FIND[B][E][V][P][C]["] del search del [,range]
Description: The FIND command finds all occurrences of a pattern within the range of the program, and displays all the lines found. 'del' is any character not in the search string or in the option characters. For an explanation of the option characters see the CHANGE command.

Command:FVER
Function: Verify a source file on disk against memory
Syntax: FVER "filename"
Description: The FVER command which works only with source files, works the same as the VERIFY command does with program files. After the file is compared, either the message ?VERIFY ERROR or VERIFY OK. is displayed.

Command:GET
Function: Read a source file into the editor
Syntax: GET "filename"
Description: The GET command loads a source file into memory, assigning arbitrary line numbers starting at 1000, in increments of 10.

Command:KEY
Function: Define a key as a special function
Syntax: KEY "key", <expr>
Description: The key command allows any key to be defined as a string. Up to 15 keys may be defined with a total length of 255 characters. None of the key functions are active when in quote mode. To clear an individual key type KEY "z","" where "z" would be the key to clear. For ease of input, quotes and return can be put in quotes as shift-q and shift-z respectively. The key character can be an expression but must start with quotes (so it won't be mixed up with the other key commands).

Example
```
KEY "&", ">$0Z"
KEY ""+chr$(131),"loadQ:*Q,8,1"
```

Command:KEYS
Function: To enable/disable the key functions
Syntax: KEYS [-]
Description: The default is enabled. To enable the key functions type KEYS and (return). To disable the key functions type KEYS- and (return).

Command:KEYC
Function: To clear all keys.
Description: To clear all the definable function keys type KEYC and press (return).

Command:KEYL
Function: List all defined keys
Description: The KEYL command will list all of the defined keys in the form KEY "x", "string".

Command:KEYR
Function: To set the repeat key timing
Syntax: KEYR <delay>, <repeat>
Description: The KEYR command will set the initial delay and repeat delay in jiffies (1/60th of a second) for the automatic repeat key. This also controls scrolling speed.

Command:KILL
Function: Disable SYSEDT
Description: The KILL command turns the SYSEDT completely off. Once the KILL command has been executed the only way to re-start SYSEDT is to re-boot in the usual manner.

Command:LIST
Function: List the program in memory
Syntax: LIST [range]
Description: The LIST command will list a basic program or text file in the same way that basic will, except that READY. and BREAK/READY. will not be displayed or printed. This allows listings to be sent to the printer without extra garbage. It also allows listing of a few lines, then scrolling up/down without READY. in the middle of it. **All of the SYSEDT commands that require a range of lines will also accept label names instead of line numbers.**

Example
LIST 100-200
LIST loop1-loop2

Command:LOADAT
Function: Loads a program file at a specified address.
Syntax: LOADAT "filename", <address>
NOTE:Address can be in decimal, hex, octal or binary. See the CONV command.
Description: The LOADAT command allows a file to be loaded at a specified address. If the file is stored in the range of $D000 to $DFFF, then it is put in the RAM under the I/O. After a file is loaded, the original and relocated locations are displayed on the screen.

Example
LOADAT "autoboot",$7000
was $0102 to $024a
is $7000 to $7148

Command:LOG
Function: Display the start/end address of a file
Syntax: LOG "filename"
Description: The LOG command displays the starting and ending address of a program file.

Example
LOG "autoboot
$0102-$024a

Command:OLD
Function: Restore the program after the NEW command.
Description: If you by accident NEW a program and want to get it back, type OLD and your program will be returned to you. The OLD command assumes that you rather run the risk of crashing the system than not get your program back. Most syntax errors between the time you enter NEW and OLD will create a variable and thus overwrite the program, causing the first few lines to return garbage.

Command:PASSWORD
Function: Password Encrypt Source Files
Syntax: Password <return>
Alpha <return>
Description: The Password command allows source files to be encrypted as a security measure. It is invoked by typing PASSWORD and pressing the RETURN key. Then you need to type in the password for the file and press RETURN. To disable the encrypt function, enter a password of nothing (Just RETURN). Be careful to remember your password, as the file can not be recovered without the password. EXEC files should not be encrypted.

The program "CPASS" will allow you to ENCRYPT or DECRYPT a group of programs while copying from one disk to another or onto the same disk.

Command:PRINTER
Function: Set the default printer device number
Syntax: PRINTER <device number>, [L]
Description: The printer command defines the default device number to be used for the * command. The printer command is to allow multiple printer operation. It is also used with the 'L' suffix to provide linefeeds if necessary.

Command:PUT/CPUT
Function: Send program text to disk as a sequential file
Syntax: PUT "filename"
CPUT "filename"
Description: The PUT/CPUT commands store to the disk drive a file of the source currently in memory. Each line is output and a carriage return is sent at the end of each line. The PUT command sends the lines verbatim. The CPUT command crunches the lines by throwing away excess spaces if they are before a ';' and not enclosed in single or double quotes. If the file exists, it will prompt REPLACE (Y/N)? Answer Y to replace file, or N to abort.

Command:RENUMBER
Function: Renumber all or part of a program
Syntax: RENUMBER <start>, <step> [,<range>]
Description: The renumber command will change the line numbers of all or part of the current program into even increments. If EDIT status is on BASIC programming then IF...THEN, IF...GOTO, GOTO, GOSUB, ON...GOTO, ON...GOSUB, LIST, RUN, and GO TO are renumbered as well. All non-existent references have an exclamation mark inserted before the old line number.

Examples
RENUMBER 1000,10
RENUMBER 1000,10,500-560
RENUMBER 1000,10,LOOPSTART-LOOPEND

Command:RESTORE
Function: Restore all editor conditions to default.
Description: The RESTORE command restores all the SYSEDT conditions to their default values.

Command:FAST
Function: Turn on Fast DOS.
Description: The FAST command will turn on the Fast DOS functions when using a 1541 or compatible disk drive. The commands affected are **/, uparrow, GET, LOG, BLOAD, BRUN.** The default setting is FAST on.

Command:SLOW
Function: Turn off Fast DOS.
Description: The SLOW command will turn off the Fast DOS functions. This may be required if the 1541 compatible disk drive is dirty or out of alignment, causing errors to occur.

TSDS Assembler (SYSASM)

Introduction

An assembler is a program that takes a source file written in a symbolic format and converts it to machine code that the 6510 processor can execute. Machine Language programming is quite different from other sorts of programming, such as BASIC because you have to instruct the processor exactly what to do, using instructions that manipulate data in its simplest form.

naturally, there are advantages to knowing how to do this. For one, programs written in machine language execute much faster than those written in any other language. In addition, there are some things that you may want to do that just aren't possible in BASIC, such as working with interrupts. The work needed to understand machine language is far outweighed by the new world that is opened up.

What is called machine language is just a string of bytes that the processor executes. This is not easily understood by a human, nor is it easily manipulated by one. To help in this task, an assembler like SYSASM is used. SYSASM lets you write a program in a symbolic fashion. You use mnemonics to represent the operations the processor can do, and by specifying addressing modes, using labels, and variables, you create more readable, understandable, and more easily modified code. The actual purpose of the assembler is too abstract your program from the actual bytes as much as possible without taking away the ability to do intimate tinkering with the hardware of the machine. As assembler takes an abstract program called **SOURCE** and turns it into a machine language program called the **OBJECT**. A line in the form **LDA #10** (Source) gets turned into **10010100100001010** (Object).

Assemblies are accomplished by typing the **ASSEMBLE** command on the screen (Assuming TSDS is loaded into the computer).

An Assembly

An actual assembly is a complicated process. The most important thing that you should know about a SYSASM assembly is that it is a two-pass assembler. This means that SYSASM reads over the Source file(s) two times. On the first pass, SYSASM determines how many bytes each statement will take up in memory. It does this so that it can determine the location of each statement and set the program counter properly. All labels are given their value on the first pass. Certain directives you will see described later only operate on the first pass.

When SYSASM reads in a statement from the source file, it goes through the following process. First it parses (splits apart) your statement into the major components. After it does this, it figures out what kind of opcode or directive the line contains. On the first pass, symbols are assigned. On the second pass, directives are worked through and the instructions decoded. Expressions are evaluated and code is generated. Finally,, a listing is given if requested. If there is an error, a listing of the line is given under any circumstances, and the code generated with a zero as the first byte (BRK). After all this, SYSASM goes on to fetch the next instruction.

At any time while SYSASM is assembling, you can press the **RUN/STOP** key to pause the assembly. After the stop key is pressed, the message **Q-QUIT, C-CONTINUE** is displayed on the screen. Pressing the **Q** key aborts the assembly, and pressing the **C** key continues the assembly.

On a long assembly, SYSASM gives you some idea of what is going on by printing any file changes under any circumstances.. Also, during the assembly, the current source line count and the current program counter are displayed on the screen.

One of the main problems with assembling files is returning back to the computer and seeing the message **ERRORS=0025** and not seeing the first 20 lines too see where the errors occurred. SYSASM solves this problem. When an error occurs, the error message and line listing is saved in memory. After an assembly, just type **LIST** (or ***LIST** for printer) to see (or print) all your error messages.. If there are more messages than will fit in 8k, then the excess messages are not saved.

An Example Simple Assembly

To see how TSDS Assembler works, try a sample program. First load in SYSEDT as explained previously. Now type NEW to delete any previous BASIC program. Now type in the following sample program.

```
 10 ;*** SAMPLE PROGRAM ***
 20 *=$1000
 30 ;
 40 LDX #0
 50 LOOP TXA
 60 STA $0400,X ;SCREEN MEMORY
 70 LDA #1
 80 STA $D800,X ;COLOUR MEMORY
 90 INX
100 BNE LOOP
110 RTS
120 .END
```

The above program fills the top portion of the screen with all of the possible character values. You can see it loading the X register with a value zero. It then stores the X register by transferring it to the accumulator, then to the start of screen, indexed by the X register. It then stores the colour white in the background colour memory, indexed by the X register. After the store it then bumps the X register with **INX**, and tests if the register has looped up to zero again. If it has not, then the loop is done again. If it has, then it returns with a **RTS**.

The following items should be noted about the listing. It is free format. There is no indentation like you might see in a normal assembler listing. The statement on line 50, which has a label called 'loop' is indented no differently from the others. Secondly the statement on line 120 begins with a dot, and looks like no assembler mnenomic in the 65xx set. What does all this mean?

One of the rules of the MOS standard is that assembler files are free form. No indentation is required to distinguish things. SYSASM determines if the line is a machine operation, a statement with a label, or one of its own directives. SYSASM knows a statement has no label if the first word on the line is one of the 56 mnenomics in the 65xx set. This means that you cannot use a mnenomic name like **LDA** for a label. If the statement begins with a word that is not a mnenomic, it is assumed that it is a label. Words that begin with dots or special symbols are expected to be directives, since a label must begin with a letter.

SYSASM Source Line Formats

SYSASM goes beyond free form. In SYSASM, you can pad your statement with spaces to make it look nicer as you please. You can put as many spaces as you like between operators, mnenomics and operands, just as you can in BASIC. You must have a space between major parts of a statement, like between the label and the opcode, and between the opcode and the operand, but otherwise you may pack things together as you please.

In the source listing, ***=** is a directive that sets the program counter. Also the line **.END** is the end of file directive. You will learn more about these later.

Now that you understand a little more about the program, you are ready to run it. Make sure you have a diskette that is ready to accept files. Type PUT "0:EXAMPLE" and press (return). This command will save the source file to the diskette. Then type ASSEMBLE "EXAMPLE" TO "EX.OBJ" and press (return). You should instantly see the assembler message come up on the screen and in a few seconds at the bottom will be the cursor flashing. If any error messages appear, you should type GET "EXAMPLE and check to make sure that you have

entered the program correctly. To execute the program type BLOAD EX.OBJ and press (return). The object program should now load into the computer's memory. Clear the screen and move the cursor down to about the middle of the screen. Now type SYS 4096 and press (return). The top portion of the screen should instantly contain lots of characters and the cursor should reappear. If this didn't happen then you must have done something wrong.

Source Line Format

Let's go into more detail on the format of a source line to SYSASM. A full source line looks like this:

LABEL OP OPERAND;COMMENT

As noted previously, the entry is free form, so typing stuff is fairly easy. On the other hand you might feel that a free format listing is difficult to read and hard to play around with. It is harder to spot the labels, instructions and comments because they do not all line up like you are used to in a listing. If you feel this way, you may pretty up your listing with proper indentation of the various parts of a line.

Labels

Any normal SYSASM statement can have a label. The label will be assigned the value of the program counter at the beginning of the assembly of that statement. Normally, you only want to label statements that generate output, like machine instructions or directives. All statements can be labelled however. The first character of a label must be a letter. The following characters may be either a letter or a digit. The maximum length of a label is thirteen characters. As noted previously, a label may not be the same as one of the 56 three letter mnenomics for the 65xx instructions. Nor should the label 'a' be used, since this letter can indicate the accumulator addressing mode.

Opcodes and Operands

After a label comes the most important part of the statement, called the op. This can be either an opcode, directive or a macro. If there was no label on the statement, this would be the first thing on the line. The op field is short. All machine opcodes are three letters long, and the directives are from one to four characters long, and macros are up to thirteen characters long.

Addressing modes

The operand of a machine instruction should usually give the

addressing mode you wish to use for that instruction. The 65xx has a wide range of addressing modes which you will find detailed in various books on the subject. Not all addressing modes can be used with all instructions, so you have to be careful what kind of instructions you use. SYSASM will tell you if you try to use an invalid combination by issuing an error message.

The syntax for the various addressing modes are given below. Since most operands include an expression with the addressing mode, the characters **EXPR** are used to indicate where an expression should go. For example, the syntax for **IMMEDIATE ADDRESSING** is given as **#EXPR**. That indicates that your statement should look something like **LDA #4**. The various modes are as follows:

Syntax	Name	Example
#EXPR	Immediate mode	LDA #IO
EXPR	Absolute or Relative mode	JSR SUBRT
A	Accumulator	ASL A
BLANK	Accumulator	ASL
EXPR,X	Absolute, indexed by X reg.	LDA TABLE,X
EXPR,Y	Absolute, indexed by Y reg.	STA TABLE2,Y
(EXPR,X)	Zero page indexed by X reg. followed by indirection.	LDA (PTAB,X)
(EXPR),Y	Zero page indexed by Y reg.	LDA (PTR),Y
(EXPR)	Indirect mode	JMP (VECTOR)

In your readings about the 65xx, you will have read about zero page addressing modes. In SYSASM you only use absolute addressing modes on your instructions. SYSASM will determine if it can use a zero page instruction in place of an absolute one, and do this if possible.

Comments

A comment may follow a statement at any point in the program. You separate a comment from the rest of the statement by a semi-colon. You need not put a space before the semi-colon if you don't desire to. In addition a line may have nothing on it but a comment. Such a line begins with a semi colon and contains plain text after it.

You now have a basic understanding of what a SYSASM source line looks like. You can look at some of the sample programs that can be found on the diskette with your copy of TSDS.

Assembler Expressions

A good part of the power of SYSASM resides in its expression evaluator. This is the part of the program that calculates the values of the instruction operands from the expressions you give. SYSASM, in helping you to abstract your program as much as possible, allows you to use complex expressions involving a wide set of operators and operand formats.

SYSASM allows you to use an expression in most places that require a value, as in BASIC. You must remember, however, that SYSASM deals with 16 bit values rather than floating point like BASIC. In addition you should note that the order of evaluation of the terms on a SYSASM expression is strictly left to right unless parenthesis are used, and does not depend on the operator involved.

Expressions consist of terms and operators. Terms are essentially values, like numbers. Operators give operations like addition and subtraction.

Term Formats

A term in a SYSASM expression can take one of the following forms:

Decimal: You can use a number just as in BASIC. The number may be anywhere from 0 to 65535. The number requires no prefix. Example: LDA 10

Hexadecimal: Many assembler programmers like to work with numbers in hex, since it is easier to cope with the binary nature of the machine. You can give a hexadecimal number by preceeding it with a **$**. You need not put any leading zeros on your number if you do not wish to, as is demanded by most machine language monitors. Example: JSR $FFD2

Octal: Octal is not a commonly used numeric base, but for compatibility with other assemblers it is included. You can give an octal number by preceding it with an **@** and like hex, leading zeros are optional. Example: EOR @377

Binary: Machines work in binary, and sometimes you wish to provide your numbers in that format. This might be the case if you had to deal with a string of bits in operations involving AND or ORA instructions. Quite often you will find that you have to deal with single bits in some of the I/O of the computer. You can enter your operand in binary by preceeding it with a **%**. Leading zeros are optional. Example: EOR %10000000

Program Counter: If desired you can use the value of the program counter as the value of a term. This can be done if you want to work with an expression relative to the location your statement will reside in memory. Use the asterisk * symbol for this. Asterisk has the value of the program counter before the current instruction is assembled. Example: BEQ *+3 ;SKIP 1 BYTE

Character: You may want to work with ASCII characters. Rather than remembering the code for **a** is 65 or $41, you can type the character surrounded in double (or single) quotes. Example : LDA #"A" or LDA #'A'

Label: As you have seen in previous examples, you can use

labels as terms. The syntax for labels is given in the section on statement syntax. Using a label means using the value assigned to it by some means. If you have a value that you can assign a symbolic name to, it is best to do so, since it will make your program easier to read. In addition, if the value changes, you need only change it in one place in the program.

Expression: You can use a complete expression as a term if you surround the expression in parenthesis. This allows you to change the order of evaluation when the value of the expression is calculated. For example: ($100+4) is a valid term. You will see more examples of this later.

Negation: You can use the negative of any term by placing a negative sign in front of it. Most commonly you wish to have negative decimal number, but you can negate labels and whole expressions in parenthesis if you want.

Lower/Higher Byte: You can use the lower half < or higher half > of a sixteen bit expression by having the less than or greater than sign respectively in front of the term. You could use a portion of a whole expression by enclosing the expression in parenthesis. Example: LDA <(lab+1+7)

Forced Absolute Addressing: As mentioned previously, SYSASM will turn the addressing mode of an instruction into a zero page mode if possible. Normally, a shift to zero page causes no harm to the program, since the operation performed by the zero page instruction is the same as that of the absolute instruction. The exception is in the case of indirect addressing modes. The zero page indexed modes pretend that the machine has only eight bits of address space - that is to say 256 bytes. This means that if you take the zero page base address 250, and index off it with, a register containing the value 20, you will address the byte at location 14 rather than location 270. You might run into the problem with a sequence like this.

```
TABLE=$0100
LDA TABLE-1,X
```

SYSASM will generate a zero page instruction here which is probably not what you want. You can force SYSASM to use absolute addressing modes by putting the ! modifier in front of the expression. In this case **LDA !TABLE-1,X** will give the desired result.

Operators: To make expressions, you combine terms with operators. SYSASM operators take a left and a right operand just like operators in BASIC.

In SYSASM the operators are:

+	Add values
-	Subtract values
*	Multiply values

```
        /     Divide values
        .     Boolean Or values
        &     Boolean And Values
    uparrow   Boolean Exclusive Or values
```

All of these operators take 16 bit values, but with certain of them it would be silly to use large values. It would be silly to multiply by a number greater than 32767. Multiplication that creates numbers that are larger than 16 bits produce an **Illegal Quantity Error.** The remaining operators don't give such errors, since overflow is often quite legitimate in the realm of two's compliment arithmetic.

The following examples show that you can combine terms with operators as you please. You should remember that a SYSASM expression is evaluated left to right as it is encountered. For example the result of **5+4*3** in SYSASM is 27, since the expression is evaluated as **(5+4)*3.** In BASIC the value would be 17. If you wanted the order that BASIC gives, use **5+(4*3)**, since the parenthesis alter the order of evaluation.

Sample Expressions: Here are some sample expressions to show some of the uses of the expression evaluator. The program doesn't do anything.

```
START   LDA TABLE-1+(TABLEN*2)
        STA $8000
        LDA #TABLE/8
        LDA #>START-1
        LDA #SIZE+>TABLE
        LDA #%1011+@3+$F
        LDA #-12
        LDA #(TABLE+2)
        LDA #<(ENTRY+TABLE+128)
        LDA #TABLE+(ENTRY*ENTSIZ)+OFF
        STA SCREEN+(LINE*COLUMN)+ACR
CODELEN=*
        LDA #>CODELEN
        LDY #<CODELEN
        JSR DOIT
MSG     .BYT "THIS IS SOME TEXT"
MSGLEN=*-MSG
```

Uses And Errors

You now should have some idea of how to form a SYSASM expression. You can use the complete power of the expression evaluator in the operand of any expression. In addition, you will read about several other places where expressions can be used. Be sure to take full advantage of this ability. A number of errors can pop up while using expressions. Check the appendix on error messages if you run into one.

Assembler Directives

Directive	Function
=	Equate label
backarrow	Redefine label
*=	Set PC
*S=	Set PC (no bytes)
.END	End of file
.BYT	Byte
.ASC	ASCII
.POK	Poke
.WOR	Words
.DBY	Double Byte
.FIL	File
.LIB	Library File
.BIN	Binary File
.SST	Save symbol table
.IFE	If zero
.IFN	If not zero
.IFP	If positive
.IFM	If minus
.ADD	Add to code
.EOR	Eor to code
.SCS	Set checksum
.CCS	Output current checksum
.LPP	Set lines per page
.PAG	Page Define title, eject page
.SKI	Skip lines
.LIS	List
.NOL	No list
.SYM	List Symbols sorted alphabetically
.SYN	List Symbols sorted numerically
.SYO	List Symbols evaluated order
.NOS	No List Symbols
.INV	Input Value
.INB	Input y/n
.PRT	Print text
.MAC	Macro Definition
.MND	Macro Definition end
.MEX	Macro Expand list
.MDX	Macro Don't Expand list

EQUATE [=]
Syntax: LABEL = <expression>
The equate directive is used to assign a value to a symbol. The operand field may contain the numeric value of the symbol or an expression evaluated by the assembler. The equate directive is only performed on Pass1.

REDEFINE [backarrow]
Syntax: LABEL backarrow <expression>

The redefine directive is used to assign a value to a symbol that has already been defined. If a re-definition is attempted using the equate directive then a re-definition error is generated. If the label is redefined with the redefine directive then no error message is generated.

SETTING THE PC [*=]
Syntax: *= <expression>

When the * referring to the assembly location counter appears on the left hand side of an equate, it is redefined to the value of the expression on the right hand side. The expression may not contain a forward reference. The location may change forward or backward as long as code has not been assembled to that point. Once code has been output, the PC may only go up, and excess space between blocks of code is filled with zero bytes.

SETTING THE SKIP CODE PC [*S=]
Syntax: *S= <expression>

This directive is the same as the 'setting the pc' directive except that when within code, the bytes skipped are not padded with bytes but instead left blank. The pc may also go in a negative direction in relation to the old pc value. The purpose for this would be a program that requires a loader because it might run in hi-memory. Example: At the beginning of the loader the directive ***=$0801** is issued then after the loader the directive ***S=$A000** is issued followed by the program in hi memory. After the object code is loaded if the loader ended at $0850, then the code which runs at $A000 would sit at $0851 and require the loader to transfer them up. This would require only one file and only one assembly to work. It could be done by creating 2 files but this is much easier. This is also useful for writing code that runs in the disk drive. At the end of the code that runs in the computer you could have ***S=$0300** and follow it with the code to run within the disk drive. Then upon execution, code that runs in the computer could download the code that runs within the disk drive.

END DIRECTIVE [.END]

The .END directive, if used, marks the end of a source program. Any statements written beyond an .END will be ignored by the assembler. The .END directive may be written with a label, however it doesn't have any operands. Use of this directive is optional but is recommended for program documentation purposes.

BYTE DIRECTIVE [.BYT]

Syntax: .BYT <expr>[,<expr>...]

The byte directive is used to reserve one byte of memory and load it with a value. The directive may contain multiple operands which will store values in consecutive bytes. ASCII strings may also be enclosed in single or double quotes.

```
Example: HERE  .BYT 2
         THERE .BYT 1, $F, %101,@7
         ASCII .BYT 'ABCDEF',"FRED"
```

Note that numbers may be represented in the most convenient form. In general, any valid expression which can be resolved to 8 bits may be included in this directive. If it is desired to have a quote within a string this may be done by putting two quotes in the string. EG.: .BYT 'JIM''S CYCLE' could be used to store JIM'S CYCLE

ASCII DIRECTIVE [.ASC]

The .ASC directive is exactly the same as the .BYT directive and is only for file conversion compatibility from other assemblers.

POKE DIRECTIVE [.POK]

The .POK directive is the same as the .BYT and .ASC directives except that the values of the text within quotes are not the ASCII value but the screen poke value.

WORD DIRECTIVE [.WOR]

Syntax: .WOR <expr>[,<expr>...]

The .WOR directive is used to reserve two bytes of data at a time (LO/HI format). Any valid expression except for text strings are valid as parameters.

```
Example: HERE  .WOR 2
         THERE .WOR 1,$FF03,@3
         WHERE .WOR HERE,THERE
```

The most common use for the .WOR directive is to generate address tables as shown in the example labelled WHERE which stores the 16 bit addresses of HERE and THERE. Addresses in the 65xx processors are fetched from memory in the order low byte, high byte, and therefore the word directive generates the values in that order. The hexadecimal portion of the second example above ($FF03) would be stored $03,$FF.

DOUBLE BYTE DIRECTIVE [.DBY]

Syntax: .DBY <expr>[,<expr>...]

The double byte directive is exactly the same as the word directive except that the values are stored in a high byte, low byte fashion.

FILE DIRECTIVE [.FIL]
Syntax: .FIL filename [,D(x)] [,S[,A]] (x=Device #)
The .FIL directive terminates assembly of the file containing it and transfers source reading to the file named in the operand. There are no restrictions on the number of files which may be linked by the file directive. The ,D ending selects the device number (none defaults to the disk device number set by the DISK command). Eg: D8, D9. The ,S ending is only valid when the current file is the control or main file. The **,S** ending tells the assembler to prompt the user to swap diskettes. Because the **,S** ending swaps diskettes, the object code will be in 2 (or more) files. The files can either be concatenated together or both loaded and run. If they are to be loaded, then the load address must be sent to the other file(s) and that is what the **,A** ending does. The file names for concurrent files are **NAME.1** , **NAME.2** etc. where NAME is the name of the object file. If no object file was selected then no object files are output.

LIBRARY FILE DIRECTIVE [.LIB]
Syntax: .LIB filename [,D(x)] (x=Device #)
The .LIB directive allows the user to construct files with commonly used subroutines. When a .LIB directive is encountered the assembler temporarily stops reading from the current file and continues reading from the file named. Processing of the original source file continues when the .END directive or an end of file status is returned from the library file. A library file called by a .LIB may not contain another .LIB but it may contain a .FIL (only 2 source files may be opened simultaneously).

Example

```
:
CONTROL          SECONDARY
.LIB file1  - ->FILE1
<- - - - - - - -.END
.LIB file2  - ->FILE2
                .FIL file3
               FILE3
<- - - - - - - -.END
.FIL file4
FILE4
.END
```

BINARY FILE [.BIN]
Syntax: .BIN filename [,D(x)][,A]
The Binary file directive is just like a big .BYT statement. The file is opened and all binary values are read in and output verbatim to the object file. Processing continues in the main file when an end of file status is returned. This is very useful when integrating screen images,character sets,sprites and instructions into your program. The **,A** ending tells the assembler to skip the first 2 bytes (load address) of the file.

SAVE SYMBOL TABLE [.SST]
Syntax: .SST filename [,D(x)]

The save symbol table directive saves the symbol table (Pass 1 only) to the filename. The format of the file is a source file and the lines are 'LABEL = $VALUE'. Each line ends in a C/R and the last line has the .END directive on it. **.SST will only save the labels up to the point in the program where it is executed.** To save all labels, make sure .SST is the last line in the program.

CONDITIONAL ASSEMBLY DIRECTIVES [.IFE, .IFN, .IFP, .IFM]
Syntax: .IFx <expr> [<]

```
.IFx
 optional
 code
 here
>
```

The conditional assembly commands evaluate the expression and if the value corresponds to what they require (=0,<>0,<32768,>=32768) then processing is continued on the next line. If the values don't correspond, then the following lines are skipped until a line with > alone is found, then processing is continued on the next line. If labels are used in the expression they must be defined in Pass 1 before the **.IFx** directive is processed.

ADD DIRECTIVE [.ADD]
Syntax: .ADD <expr>

The .ADD directive adds the value in the expression to all of the bytes output to the object file until the next .ADD directive. The default is .ADD 0.

EXCLUSIVE OR DIRECTIVE [.EOR]
Syntax: .EOR <expr>

The .EOR directive exclusive ORs the value in the expression with all of the bytes output to the object file until the next .EOR directive.

SET CHECKSUM [.SCS]
Syntax: .SCS <expr>

The set checksum command is used to reset the current checksum value. The checksum initializes to 0 and every byte output to the object file is exclusive OR'ed with the previous checksum. The main purpose for the checksum command is so that the program can see if it has been modified, as in protection schemes or self testing procedures. The current checksum value will be output with the .CCS directive as 1 byte. Using a program loop, the checksum can be compared to the real checksum.

OUTPUT CURRENT CHECKSUM [.CCS]
Syntax: .CCS

The current checksum directive outputs the current checksum (1 byte) to the object code. See .SCS set checksum directive.

SET LINES PER PAGE [.LPP]
Syntax: .LPP <number>

For assembly output to a printer, the lines are automatically paged by the assembler. The page format is one blank line followed by the page count and title then followed by one blank line. Following that is the source lines and then three blank lines at the bottom of the page. The lines per page defaults to 66 lines. Because the page is broken up into three lines at the top and three lines at the bottom, the number of source lines per page is the number set minus six. Using the default as an example, the page would consist of three lines (header), then sixty lines of code, then three lines (footer).

PAGE DEFINE/EJECT [.PAG]
Syntax: .PAG [title]

At the top of each page is the name of the page and the page count. The page name defaults to the name of the source control file. The .PAG command with no parameters ejects the paper until the next page and then continues the printout. The .PAG command with a parameter ejects the paper to the top of the next page and redefines the page title name to the one given. The maximum length of the page title name is twenty characters.

SKIP LINES [.SKI]
Syntax: .SKI [number]

The skip directive sends to the printer the parameter number of blank lines. If no parameter is present then a 1 is assumed. This is useful for separating subroutines in the listing without wasting a page for each one.

LIST [.LIS]
Syntax: .LIS

The .LIS directive tells the assembler to send the listing to the printer (if selected). Output is sent to the printer until a .NOL directive is encountered.

NO LIST [.NOL]
Syntax: .NOL

The .NOL directive suspends output to the printer. The list and nolist directives are used mainly so that if desired, only portions of the output will be printed. The .NOL directive remains in effect until a .LIS directive is encountered.

SYMBOL LIST ALPHABETICALLY SORTED [.SYM]
Syntax: .SYM

The symbol list directive outputs the symbol table (sorted alphabetically) to the printer at the end of Pass 2. **This is the default setting.**

SYMBOL LIST NUMERICALLY SORTED [.SYN]
Syntax: .SYN

The symbol list directive outputs the symbol table (sorted numerically) to the printer at the end of Pass 2.

SYMBOL LIST NOT SORTED [.SYO]
Syntax: .SYO

The symbol list define order directive outputs the symbol table, in the order that it was created, to the printer at the end of Pass 2.

NO SYMBOL LIST [.NOS]
Syntax: .NOS

The symbol table is not sent to the printer.

INPUT VALUES [.INV]
Syntax: .INV label

The input value directive inputs a value from the user during pass1 only. The value may be in hex, decimal, binary, octal or ASCII. If the value input is in error then the command is re-done until the entry is valid. See example in .PRT section below.

INPUT BOOLEAN [.INB]
Syntax: .INB label

The input boolean directive inputs a 'Y' or 'N' from the user and returns a value to the label 1=YES, 0=NO (Pass 1 only). If no character is input then a 'Y' is assumed. See example in the .PRT section below.

PRINT [.PRT]
Syntax: .PRT "text...."

The print text directive prints the text in quotes to the screen (Pass1 only). A RETURN is then printed after the text.

```
EXAMPLE:    .PRT 'Input type of computer ?'
            .PRT '1 = Commodore 64'
            .PRT '2 = VIC 20'
            .PRT '3 = PET BASIC 2,3'
            .PRT '4 = PET BASIC 4'
            .INV BASIC   ;input value
            ;
            .PRT 'Eighty columns?(y/n)'
            .INB EIGHTY  ;input boolean
```

The previous would enable one source file to be able to assemble a program with versions for different machines (or whatever) without having a different control file for each one. The source, using the .IFx directives can then patch different code for each machine where required.

MACRO DEFINITION [.MAC]
Syntax: .MAC macroname

The .MAC directive is used to allow the definition of macros. A macro must be defined before it can be used or defined by another macro. The lines of text after the .MAC directive are stored in the macro definition area until the directive .MND is encountered.

MACRO DEFINITION END [.MND]
Syntax: .MND

The .MND directive is used to mark the end of a macro definition. The line may also contain a label which will be part of the current macro. See the .MAC directive.

MACRO LIST EXPAND [.MEX]
Syntax: .MEX

The .MEX directive tells the assembler to output the macro, line for line, to the printer (until a .MDX is encountered).

MACRO DON'T EXPAND [.MDX]
Syntax: .MDX

The .MDX directive tells the assembler that when a macro is encountered to only list the macroname and the parameters to the printer. This is in effect until an .MEX is encountered.

Macro Capabilities

A macro is essentially a facility in which one line of source code can represent a function consisting of many instruction sequences. For example the 65xx instruction set does not have an instruction to increment a double byte memory location. A macro could be written to perform this operation and represented as INCD. This macro would appear in your assembly listing in the mnemonic field similar to the following:

```
        STA TEMP
        INCD TEMP1 :;double byte inc macro
SKIP    LDA TEMP2
```

Before a macro can be used it must be defined. A macro name may not be the same as a 65xx mnemonic or a defined label. The syntax for a macro name is the same as the syntax for a label. Macros take the general form shown in the following code:

```
        .MAC <macro name>
         text of macro
[label] .MND
```

The directive .MAC defines a macro with the given macro name. The user does not explicitly declare parameters. The macro may contain arbitrary text, but a macro cannot be defined within a macro.

The assembler will not produce any object code until the macro is called. To call a macro, the user simply gives the macroname and lists the parameters as indicated in the call line below:

```
MACRONAME param1:param2:...
```

The macro name must be delimited by a space from the first parameter as indicated above.

Within the text of the macro definition, a parameter is designated by the temporary symbol **?** followed by a number. Thus **?3** designates parameter 3. The maximum number of parameters passed to a macro is limited to the size of the source line. During assembly, if a macro call is encountered, the text included in that macro is inserted into the assembly at that point and the parameters at the calling point are substituted for the temporary names. If the user fails to supply a parameter when the macro is called, the assembler will substitute a null string for that parameter. Labels defined within macros are assumed to be local to that macro. If global labels are to be defined within a macro use the **(uparrow)** character before the label name.

Macro Parameter passing

Parameters are separated by colons. If a colon is found within a parameter enclosed in single or double quotes then it is assumed that the colon is not an end of parameter marker. Empty parameters in call lines are denoted by colons with no text between them.

```
FUNCTN aa:::dd: ;parms 2,3 are empty
FUNCT ::cc::ee: ;parms1,2,4 are empty
```

A macro parameter is supplied to a macro without leading or trailing blanks, so parameters can pass addressing modes, expressions, label names, text or almost anything.

Example legal parameters:

```
ADD (ptr),y:#$02:          ;add 2 to (ptr),y
OPEN 15:8:15:"i0":         ;open15,8,15,"i0"
PRINT 147,13,13,'menu',13: ;print text
```

If a comment is desired following the macro parameters then seperate the semi-colon and the last parameter by a colon. Otherwise the comment is assumed to be part of the last parameter.

Macro Examples

```
        .MAC PRINT
        LDY #0
PRINTL  LDA PRINTT,Y
        BEQ PRINTF
        JSR CHROUT
        INY
        BNE PRINTL
PRINTT  .BYT ?1,0 ;passed text goes here
PRINTF  .MND ;done here
```

PRINT 147,13,'title':;call print macro

```
        .MAC OPEN
        LDA #?1
        LDX #?2
        LDY #?3
        JSR SETLFS ;global label
        LDA #OPENNF-OPENNS ;len of name
        LDY #>OPENNS
        LDX #<OPENNS
        JSR SETNAM
        JSR ROPEN ;rom's open $ffc0
        JMP OPENNF ;finished.
OPENNS  .BYT ?4
OPENNF  .MND
```

OPEN 15:8:15:"I0":;intlz drive

Assembler Error Messages

Non Fatal error messages
(assembly is continued)

```
*UNDEF LABEL*                (label hasn't been defined)
*REDEF ERROR*                (label has already been defined)
*ILL OPCODE*                 (invalid opcode name)
*ILL DIRECTIVE*              (not a valid directive)
*ILL ADDR MODE*              (not a valid addressing mode)
*ADDR MODE MISMATCH*         (don't match.  ex: jsr ($ff),y )
*ILL QUANTITY*               (too large a value.  ex:.byt 256)
*LINE TOO LONG*              (line > 80 characters)
*LABEL TOO LONG*             (label > 13 characters)
*ILL START CHAR*             (line's first character is bad)
*OUT OF RANGE*               (branch >127 or <-128 )
*MISSING COMMA*              (comma was expected)
*UNBALANCED PAREN*           (missing right bracket(s))
```

Fatal error messages
(assembly is aborted)

```
*NO END OF .IFX ERR*   (end of file encountered before end of .if)
*DISK ERR*             (I/O error)
*PC REVERSAL*          (PC has reversed within code)
*SYMBOL TABLE OVERFLOW*    (out of symbol table memory)
*MACRO MEMORY OVERFLOW*    (out of macro memory)
*MACRO STACK OVERFLOW*     (macro's nested too deep)
```

Non Fatal Error Messages
(assembly is continued)

Undefined label
This error is generated on the second pass if the label in an expression has not been defined.

Redefinition error
This error is generated if a label is attempted to be defined by the equate directive after it is already in the symbol table. If the label is truly needed to be redefined then use the redefine directive.

Illegal op-code
The assembler searches the line until it finds the first non-blank character string. If this string is not one of the valid op-codes or macros it assumes that it is a label and places it into the symbol table. However if a second field is found it is assumed to be the op-code (since only one label is allowed per line). If this character string is not a valid op-code then the error message is printed.

Illegal Directive
All assembler directives begin with a period. If a period is the first character in a non-blank field the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed.

Illegal Address Mode
If the op-code string is found to be valid and the addressing mode is found to be valid but the two together don't create a legal instruction, then the following error message is printed out. Example: jsr ($0090). In the example the jump op-code is the only op-code that allows indirect addressing.

Illegal Quantity
The value of an expression is too large. Example: **.BYT 257** In the example the value is greater than the legal byte range of 0 to 255.

Line too Long
The line read in by the assembler is greater than eighty characters in length. Try breaking it up into segments.

Label too long
All labels are limited to thirteen characters in length. When parsing, the assembler looks for a non-alphanumeric character to find the end of a label. If the length of the label found is greater than thirteen characters in length, then the error message will be printed.

Illegal Start Character

The first character on a line must be either (a-z) indicating a label or an op-code, or '.' indicating a directive or a '*' indicating the PC or a '>' indicating the end of an .if directive. Any other leading character will cause the error message to be printed.

Branch out of range

All of the branch instructions are assembled into 2 bytes of code. One byte is for the op-code and the other is for the address to branch to. To allow forward or backward branches to occur, the branch is taken relative to the position of the instruction. If the value of the byte is 0-127 then the branch is forward. If the value of the byte is 128-255 then the branch is backward. Therefore a branch can only branch forward or backward 127 bytes. If an attempt is made to branch further than these limits then the error message is printed.

Missing comma

A comma was expected but another character was found.

Unbalanced parenthesis

An expression has an uneven number of parenthesis.
Example .byt (5*7+(12/2)

Fatal Error Messages
(assembly is aborted)

***No End of .IFx**

While processing an '.if' directive, if the result of the expression is found to be false, the assembler skips the following lines until a line containing '>' is found indicating an end of .if condition. If the end of file status is generated before the '>' line is found then the error message is printed.

Disk Error

If while reading in source code or writing out object code, an I/O error is generated by the operating system then the current file name is displayed on the screen, along with the disk status message and the above error message.

PC Reversal

Because output of the object code is going to a program file, the value of the program counter can only go in one direction - up.. If the value of the program counter is attempted to go below its current value, after code has started being output, then the error message is printed.

Symbol Table overflow

The symbols required in the program are stored internally with their values. If the memory available for symbols is overflowed, then the error message is displayed. If this error is displayed, then either break the program into 2 (or more) programs, remove un-necessary labels, shorten the labels, or buy a larger computer..

Macro memory overflow

Macro memory is 8k. Macros are stored in the macro memory sequentially in their text form. If the memory is all used up, then the error is generated. If this happens, try removing all excess spaces and comments from macros.

Macro Stack Overflow

The only limit to the nesting of macros is the size of the macro stack. Each time a macro calls another macro, the macro's name, depth and parameters are all saved on a stack. Therefore the number and length of the parameters are the real macro nesting limit. If the space required to save the current macro's values is not available in the macro stack, then the error message is printed.

TSDS Sprite Editor [SPED]

With this sprite editor you can edit up to 256 sprites at once. Sprites are stored as program files. When loading a sprite file, it doesn't matter what VIC-chip block it's saved at because the editor relocates it to the Block 1. And when saving a block of sprites, you input the block number, and the file's load address is altered accordingly.

Editing Keys

All cursor editing is done with the CRSR keys or joystick number 2

```
RVS        =    Reverse sprite
CLR        =    Erase sprite
HOME       =    Home cursor
RETURN     =    Next line, Left Side
(period)   =    Draw Dot
(space)    =    Erase Dot *
back arrow=     Toggle bit *
(fire)     =    Either (set, clr, tog) depending on which
                mode is selected.  *
NOTE : * = in multicolour mode these increment the colour
value (0-3)
```

Commands from Main Menu

```
+    Go to next sprite page
-    Go to previous sprite page
g    Goto input sprite page
a    Animate mode
A    Animate mode off
c    Colour change menu
f    Block fill (with rotate)
i    Interlace
j    Joystick fire mode
k    Keyboard auto CRSR toggle
l    Load file
m    Toggle multicolour mode
o    Overlay mode
O    overlay mode off
p    Sprite push menu
s    Save sprite range
t    Transfer range
v    View sprites 0-63,64-127,128-191,192-255
z    Set spin coordinate center
```

Animate

Animate mode is used for displaying more than one sprite in the same screen location. For example: A set of sprites which look like a bird flying when flipped at a constant rate. Sprites must be in sequential order. There is two modes for displaying: uni-directional and bi-directional mode. The speed value is the delay between flipping to the next sprite. Each speed unit is in jiffies (1/60 th of a second).
IE. If the sprite range was (s=0,e=5)
Unidirectional: 0,1,2,3,4,5,0,1,2,3,4,5,0,1,2,...
Bidirectional: 0,1,2,3,4,5,4,3,2,1,0,1,2,3,4,...

Colour Change Menu

In the menu pressing the [f1] key returns to the main menu. The colours that can be changed are the SPRITE COLOUR, BACKGROUND COLOUR, BORDER COLOUR, SPRITE MULTICOLOUR 0, and SPRITE MULTICOLOUR 1. To change the colour for the desired location press the first letter in the word, then type a colour key (CTRL 1-8) or (Commodore 1-8)
Example: You want to change the edge colour to white.
Press E then **CTRL 1**

Block Fill

The Fill command makes copies of a sprite into a range of sprites. For example if you want to make 5 fish but each slightly different, then after drawing the first in sprite number 0, you would use the fill command and enter ORIGINAL SPRITE=0, START=1, END=4, ROTATE=0. Then modify sprites 1-4 as you choose. You can optionally rotate each sprite by any number of degrees. Just enter a degree rotation when asked.

Interlace

The Interlace command is for merging two sprites together. All that need be input is the number of the sprite to be merged with the current viewed sprite.

Joystick Fire Mode

The fire button on the joystick can either toggle or clr or set the bit at the current cursor position. To select the mode just type the first character in the mode name.
Example: If you want Set mode then just type **s**

Keyboard Auto Cursor Move

When editing the sprite the cursor can either stay in the same position it was previously at, or move one character to the right after an edit key (back arrow,spc,. (or fire)) is pressed. To change the mode (move,stay) pressing the **k** key from the main menu will toggle it's status.

Load

The Load command displays a directory of all the program files on the disk based on the search pattern input and lets you choose which one to load. If the file name is greater than ten characters then only the first ten are displayed on the screen. To load a file just input the directory pattern (*=all) and crsr up/down to select the one to load. Press return when the rvs bar is on the desired file.

Overlay

The Overlay command is used for displaying more than just the current sprite being edited on the screen. It is mainly used for objects that are made up of many sprites. There are up to six overlays on the screen at a time. To make an overlay just input the overlay number (1-6) and the sprite number. Then position the sprite on the screen with the crsr keys (or joystick) and press [f1] when done.

Overlay mode off

Just input the overlay (1-6) to turn off and press return.

Push Menu

The Push menu is used for moving a sprite around. You will note that the up, down, left, right scroll commands have two different ways of scrolling. The capitols (UDLR) wrap the sprite around and the lowercase (udlr) throw away the parts of the sprite that go off the edges.

Commands

U/u	scroll up
D/d	scroll down
L/l	scroll left
R/r	scroll right
x	x mirror
y	y mirror
s	spin (1-359) degrees
[f1]	Main Menu

Note: Although the sprite is 24 x 21, a 24 x 24 buffer is preserved until the Push menu is exited.

Save

To save a block of sprites just input the filename (max9 chrs). It automatically puts "0:" in front of the name. Then input the vic chip block (0-3), starting sprite number, ending sprite number. (Note: It saves up to and including the last sprite)

Transfer

To Transfer a block of sprites just input the starting sprite number and ending sprite number of its original place, and then the starting sprite number of the destination.

View

View allows you to see up to 64 sprites on the screen at one time. Just press **v** and then the block number (1,2,3 or 4). Press any key to go back to menu. Press [f1] at menu to return to Main Menu.

Unassembler Instructions

The unassembler is a program that will attempt to change a Machine Language program back into source code. It has some problems though, with things like TEXT, DATA TABLES, JUMP TABLES, and such. This unassembler will ask you to tell it where things like that are. To determine where these things are, you must use a program like SUPERMON or EXTRAMON to disassemble the program and figure out where these things are. Once you have determined the locations of the tables, etc, write down all the start and end locations, and the you are ready to Unassemble a program.

Once the unassembler is loaded, it will ask you for the file name to unassemble. Type in the name and press <return>.

Now it will ask for the information on the tables, etc. Lets assume we have a program with the following specifications:

Type of Information	Start Location	End Location
Code	C000	C080
Bytes	C081	C090
Text	C091	C230
Words	C5E3	C5FF

Now you would just enter the above information into the program. The program will then ask for the OUTPUT FILE NAME. Give it a name different from the program you are unassembling. Now it will ask for the number of blocks to make each file. Since it may make numerous files, if a small number is chosen, or one large file is a large number is chosen, this is up to you.

Now the program will unassemble the program, and will leave you with the following files on the disk. They are (Assume output file name is [name]).

NAME	File to start assembly from
NAME.LAB	A file with all the labels and values.
NAME.00	First file of source code
NAME.nn	Last file of source code.

The unassembler will change all memory locations used (Like $FFD2) to a label in the form (ZFFD2). In fact, it just puts a 'Z' in front of the hexadecimal value.

Now, using TSDS, you can edit the program to suit your needs, and if unassembled properly, re-assemble it with your modifications.

If you would like to change all the Zxxxx labels to something more understandable, follow this procedure:

1) Load up TSDS.
2) GET first unassembled file (Including .LAB file)
3) Put TSDS disk back in and type **EXEC LABELS**
4) Now PUT the file back onto its disk.
5) Repeat steps 2-4 for each file to be changed.

What this program does is create a bunch of CHANGE commands to change things like ZFFD2 to CHROUT (It's real name.)

Converter Instructions

The converter program is used to convert source code files from other assemblers like PAL and MAE for use with TSDS. It is very simple to operate, and does a fairly good job of conversions.

Upon loading the convertor named "CONVERTER", RUN it and follow the on-screen instructions.

TSDS Sound Effects Editor [SED]

Overview

Thanks to Shelly Kantrow and David Miller (authors of "3001, a Sound Odessey") for the original sound effects play routine.

With this sound effects editor you can edit up to 256 effects at once. Effects are stored as 2 program files. The first file 'e.name' contains the data (13 bytes) for each effect, the second file 'n.name' contains the names for the effects (16 bytes) each. When loading an effects file, it automatically starts loading the effects at effects position number zero.

This effects editor is centered around a machine language effects play routine. The source code for the routine is on the TSDS diskette. The editor allows you to create sound effects and then in conjunction with the source code for the effects play routine provided, you can add sound effects to your own programs. The effects player plays up to three effects at one. If all three voices are busy playing effects when an effect is called to be played then it is ignored.

Effect Structure

Each Effect consists of 13 bytes of data. The following information is used in normal sound generation in the SID chip:

1) Frequency
2) Pulse Width
3) Attack
4) Sustain
5) Decay
6) Release
7) Waveform Type

In addition to this information, the SED also uses an effect information block consisting of REPEAT (0-15), Envelope ADSR and Intensity.

The Envelope ADSR is for changing the frequency of the effect. The intensity value is for setting the intensity of the envelope follower. The higher the intensity, the more the frequency changes in relation to the envelope follower ADSR.

Editing Keys

All Effect editing is done with the CRSR keys

> or .	=	Add 1 to frequency or pulse
< or ,	=	Subtract 1 from frequency or pulse
p	=	Play Sound Effect
n	=	Name Sound Effect
+	=	Next Effect
-	=	Previous Effect
g	=	Goto Effect
l	=	Load Effect files
s	=	Save Effect files

Play Effect
The Play Effect command plays the currently viewed effect. If no noise is output then check carefully to make sure that the proper frequency is set. Also if the effect is in pulse mode then make sure that the pulse width is large enough.

Name Effect
The Name sound effect command allows all of the effects to have a 16 character name associated with them. This allows you to remember what effects go where while developing them.

Load
The Load command displays a directory of all the sound effects files on the disk based on the search pattern input and lets you choose which one to load. To load a file just input the directory pattern (*=all) and crsr up/down to select the one to load. Press <return> when the bar is on the desired file.

Save
To save a block of effects just input the filename. It automatically puts "0:" in front of the name. Then input the starting effect number, ending effect number. (Note: It saves up to and including the last effect)

How to play effects from YOUR program

The source code for the sound effects player is on the TSDS diskette. To add your effects to the source code do the following. load TSDS Sysedt. type **GET SNDEFFS.S. At line number 1480 there is a .BIN File Command.** Change the name in the file to your own. Then **PUT** the file to your own diskette and assemble it to object code. To test the effects program, bload the object file. Type **EDIT B** and **NEW.** Then write a test program in BASIC. First initialize the effects routine with **SYS 49152.** Then to play an effect use **POKE 49152+4,(effect number)** and **SYS 49152+3.** This should play the effects. If you wish to add the effects to your own assembler program then to setup the effects use a **JSR INITSID.** And to call an effect use a **LDA #effect number** and **JSR PLAYEFF.** If you want to add it to your own source, make sure the label **ZPSND'** doesn't conflict with any of yours and if you are using your own 60 hz interrupt code then you can call **EFFIRQ** directly yourself in your interrupt code, and delete the IRQ Vector setup that is within the **INITSID** subroutine.

TSDS Matte/Chrset Editor Instructions [Macsed]

Macsed resides in memory from $0801 to $2FFF. The two character sets reside from $3000-$3FFF. Then from $4000-$49FF are variables and finally the editing matte resides in $4A00-$7800 and the matte color resides in $D100-$FF00. The Sysedt/Sysasm editor/assembler may be resident while using Macsed.

Macsed is a program that allows you to edit character sets and then draw pictures with the character set. The main purpose for Macsed was to design screens for levels within games, although it's uses are endless. The program starts up in the Matte Editor. Pressing the 'F7' key goes to the main menu of the Matte Editor. From the Matte Editor menu you can go to the Character Set Editor by pressing 'F1'.

Matte Editor

The Matte Editor allows editing with the cursor keys. Across the top of the screen is a cross reference line between the normal character set and the one currently loaded. The Matte Editor stores mattes as 2 program files. The first file is named "B.NAME" and consists of the Number of bytes across, number of bytes down followed by the character codes. The second file "B/NAME" has the same two first bytes but the rest of the file is the color information.

Matte Editor Edit Keys

(crsr up)	Move cursor up
(crsr down)	Move cursor down
(crsr left)	Move cursor left
(crsr right)	Move cursor right
(crsr home)	Top of View screen
(crsr home x2)	Top of Matte
(insert)	Insert a character
(delete)	Delete a character
(color keys)	Select color for drawing
(return)	Move to left edge of matte
(shift-return)	Move to left edge of view screen
(F1)	Scroll cross reference line left
(F3)	Scroll cross reference line right
(F5)	Increment Background Color
(F7)	Goto Matte Menu

Matte Editor Menu

(F1)	Goto Character Set Editor
(l)	Load a Matte
(s)	Save a Matte
(r)	Set a Range
(c)	Copy a Range
(t)	Transfer a Range
(e)	Erase All/Range
(b)	Blank Define
(w)	Set Window Size
(insert)	Insert 1 line
(delete)	Delete 1 line

Load Matte
The Load Matte command prompts for a load pattern. The load command then loads a directory from the disk using the pattern input. Then use the cursor up/down keys to select the file to load. The load command then asks if you want to set the Matte size to the size of the file. If you choose no. Then it asks if you want to load the matte at the cursor position. If then loads the Matte into the Matte Editor.

Save Matte
The save matte command prompts for the name of the Matte and then it prompts for **ALL OR RANGE.** Pressing a saves the whole matte and pressing 'r' only saves the range set by the range command. Note: The filename 'name' is converted to '0:b.name' and if you desire to replace and old file the name '@name' is automatically converted to '@0:b.name'.

Set Range
The set range command is used for setting a range of the Matte for the Save, Erase, Transfer and Copy commands. The cursor appears and then use the cursor up/down/left/right keys to set the range and then press return. The characters in the range are inverted on the screen while setting the range, depending on what your character set looks like they may look weird.

Copy/Transfer Range
The Copy/Transfer commands are the same except that the transfer command erases the old range. The Copy/Transfer command copies the range set with the range command to the current cursor position.

Erase All/Range
The Erase command will erase the whole Matte or Only a range set by the Range Command.

Blank Define
The Blank Define Command allows you to define the character that is used in insert and delete and erasing. This allows the space character to have a different value in your character sets.

Set Window Size
The set window size command allows you to change the size of the matte. The maximum across is 255 characters and the maximum down is 255 characters, but because of the size of the memory available the total size is a maximum of 11776 characters. After inputting the number of characters across the maximum down value is displayed on the screen.

Insert/Delete Line
The insert/delete command inserts or deletes one whole line at the cursor down line on the matte.

Character Set Editor
The Character Set editor has no Main Menu. All commands may me pressed directly just like in the Sprite Editor (SPED). Pressing the 'F7' key returns to the Matte editor. Pressing the return key allows you to edit current character. When the character sets are saved a filename like 'name' has a prefix added to it to become 'c.name'.

Character Set Editor Keys

(crsr keys)	Select a character
(color keys)	Display Character in Color
(+) (-)	Select Block
(return)	Edit Character
(l)	Load a Character Set
(s)	Save a Character Set
(r)	Set a Range
(c) (C)	Copy a Range
(t) (T)	Transfer a Range
(m)	Multi Color On/Off
(1) (2)	Select Character Set 1,2
(F1)	Scroll Cross Reference Line Left
(F3)	Scroll Cross Reference Line Right
(F5)	Increment Background colour
(F7)	Goto Matte Editor
(Commodore C)	Sset Colour of Range

Load a Character Set
The Load Character Set command prompts for a pattern, then displays the directory entries that match that pattern. You the select the character set to load with the cursor up/down keys and press return to load it. It loads into the current character set position selected.

Save a Character Set
The Save character Set command asks for a file name and then asks **ALL OR RANGE?.** Pressing 'a' saves the whole character set and pressing the 'r' key saves a range of characters. Note: When a filename like 'name' is input it is automatically converted to 'c.name'. If it is desired to replace a file then input '@name' because it is converted to '@0:c.name' automatically.

Set A Range
The range Set command is used for the Copy and Transfer Commands. Use the cursor and '+,-' keys to select the range of characters. The characters selected will appear in a different color. Press return when the range is correct. Note: The range command also remembers which set (1,2) was used and therefore allows transfer and copy between character sets.

Copy and Transfer
The copy and transfer commands work the same except that the transfer command erases the original range. They copy the range defined to the cursor position. If the shifted C and T command is used then the characters copied are inverted first.

Multi Color On/Off
The multi color command asks whether to put multi color mode on/off. If you selected 'on' then it also prompts for the 2 multi colors as well. Note: To See the character set in multicolor mode on the screen you must display it in one of the upper eight color values.

Select Character Set
Pressing the 1 or 2 key selects the first or second character set.

Set Colour of Characters in Range
Pressing the Commodore and C keys sets the colour of the characters defined in the range to the last colour used.

Editing a Character
The following keys are used to edit the currently selected character.

Character Editing Keys

(crsr up)	Cursor Up
(crsr down)	Cursor Down
(crsr left)	Cursor Left
(crsr right)	Cursor Right
(backarrow)	Oops!!
(space)	Clear bit
(.)	Set bit
(1) (2) (3)	Set Colors (Mcm Mode only).
(insert)	Insert 1 bit
(Delete)	Delete 1 bit
(u)	Scroll up
(d)	Scroll down
(l)	Scroll left
(r)	Scroll right
(I)	Insert 1 line
(D)	Delete 1 line
(s)	Spin 90 Degrees
(return)	Done Editing

C128 Changes/Enhancement

The following paragraph describes the changes and enhancements made in the C128 version of TSDS.

1) The commands **FAST** and **SLOW** have been removed to allow the resident C128 commands FAST and SLOW to function. All file transfers run at normal DOS speed.

2) Sprite Editor, Matte Editor, Sound Effects Editor, and CPASS must all be run in C64 mode only.

3) TSDS-128 is loaded by BOOTING the system. See loading instructions at the beginning of the manual.

4) The ASM or ASSEMBLE commands take on new syntax.

ASM	Assemble from source in memory. No object produced.
ASM TO	Assemble from source in memory. Object placed in Bank 0 memory.
ASM "name"	Assemble from source on disk. No object produced.
ASM "name" TO	Assemble from source on disk. Object placed in Bank 0 memory.
ASM TO "name"	Assemble from source in memory. Object placed in disk file "name".
ASM "name" TO "name"	Assemble from source on disk. Object placed in disk file "name".

5) Areas of memory used by TSDS-128.

Bank 0	$0C00 - $1000	Program scratchpad memory.
Bank 1	$1000 - $57FF	TSDS-128 Program
Bank 1	$5800 - $FEFF	Symbol Tables, Macros, etc.

6) To get error listing after assembling, you must GET the error file.
This file is located in the RAMDISK. It is loaded using the following.

GET "ERRORS"